(/)

Check If a String Is Numeric in Java

Last updated: August 10, 2023

> Written by: baeldung (https://www.baeldung.com/author/baeldung)

Java (https://www.baeldung.com/category/java) +

Java String (https://www.baeldung.com/tag/java-string)

1. Introduction

Oftentimes while operating upon *String*s, we need to figure out whether a *String* is a valid number or not.

In this tutorial, we'll explore multiple ways to detect if the given *String* is numeric, first using plain Java, then regular expressions, and finally by using external libraries.

Once we're done discussing various implementations, we'll use benchmarks to get an idea of which methods are optimal.

Further reading:

Java String Conversions (/java-string-conversions)

Quick and practical examples focused on converting String objects to different data types in Java.

Read more (/java-string-conversions) →

A Guide To Java Regular Expressions API (/regular-expressions-java)

A practical guide to Regular Expressions API in Java.

Read more (/regular-expressions-java) →

Understanding the NumberFormatException in Java

(/java-number-format-exception)

Learn the various causes of NumberFormatException in Java and some best practices for avoiding it.

Read more (/java-number-format-exception) →

2. Prerequisites

Let's start with some prerequisites before we head on to the main content.

In the latter part of this article, we'll be using Apache Commons external library to add its dependency in our *pom.xml*:


```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.12.0</version>
</dependency>
```

The latest version of this library can be found on Maven Central (https://search.maven.org/classic/#search%7Cga%7C1%7Cg%3A%22org.apache.commons%22%20AND%20a%3A%22commons-lang3%22).

3. Using Plain Java

Perhaps the easiest and the most reliable way to check whether a *String* is numeric or not is by parsing it using Java's built-in methods:

1. *Integer.parseInt(String)*
2. *Float.parseFloat(String)*
3. *Double.parseDouble(String)*
4. *Long.parseLong(String)*
5. *new BigInteger(String)*

If these methods don't throw any *NumberFormatException (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/NumberFormatException.html)*, then it means that the parsing was successful and the *String* is numeric:

```
public static boolean isNumeric(String strNum) {
    if (strNum == null) {
        return false;
    }
    try {
        double d = Double.parseDouble(strNum);
    } catch (NumberFormatException nfe) {
        return false;
    }
    return true;
}
```

Let's see this method in action:

```
assertThat(isNumeric("22")).isTrue();
assertThat(isNumeric("5.05")).isTrue();
assertThat(isNumeric("-200")).isTrue();
assertThat(isNumeric("10.0d")).isTrue();
assertThat(isNumeric("   22   ")).isTrue();

assertThat(isNumeric(null)).isFalse();
assertThat(isNumeric("")).isFalse();
assertThat(isNumeric("abc")).isFalse();
```

In our *isNumeric()* method, we're just checking for values that are of type *Double*; however, we can also modify this method to check for *Integer*, *Float*, *Long*, and large numbers by using any of the parse methods that we enlisted earlier.

4. Using Regular Expressions

Now let's use regex *-?\d+(\.\d+)?* to match numeric *Strings* consisting of the positive or negative integer and floats.

It goes without saying that we can definitely modify this regex to identify and handle a wide range of rules. Here, we'll keep it simple.

Let's break down this regex and see how it works:

- *-?* – this part identifies if the given number is negative, the dash "–" searches for dash literally and the question mark "*?*" marks its presence as an optional one
- *\d+* – this searches for one or more digits
- *(\.\d+)?* – this part of regex is to identify float numbers. Here we're searching for one or more digits followed by a period. The question mark, in the end, signifies that this complete group is optional.

Regular expressions are a very broad topic. To get a brief overview, check our tutorial on the Java regular expressions API (https://baeldung.com/regular-expressions-java).

For now, let's create a method using the above regular expression:

```
private Pattern pattern = Pattern.compile("-?\\d+(\\.\\d+)?");

public boolean isNumeric(String strNum) {
    if (strNum == null) {
        return false;
    }
    return pattern.matcher(strNum).matches();
}
```

Now let's look at some assertions for the above method:

```
assertThat(isNumeric("22")).isTrue();
assertThat(isNumeric("5.05")).isTrue();
assertThat(isNumeric("-200")).isTrue();

assertThat(isNumeric(null)).isFalse();
assertThat(isNumeric("abc")).isFalse();
```


5. Using Apache Commons

In this section, we'll discuss various methods available in the Apache Commons library.

5.1. *NumberUtils.isCreatable(String)*

NumberUtils (https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/math/NumberUtils.html) from Apache Commons provides a static method *NumberUtils.isCreatable(String) (https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/math/NumberUtils.html#isCreatable-java.lang.String-)*, which checks whether a *String* is a valid Java number or not.

This method accepts:

1. Hexadecimal numbers starting with 0x or 0X
2. Octal numbers starting with a leading 0
3. Scientific notation (for example 1.05e-10)
4. Numbers marked with a type qualifier (for example 1L or 2.2d)

If the supplied string is *null* or *empty/blank*, then it's not considered a number and the method will return *false*.

Let's run some tests using this method:

```
assertThat(NumberUtils.isCreatable("22")).isTrue();
assertThat(NumberUtils.isCreatable("5.05")).isTrue();
assertThat(NumberUtils.isCreatable("-200")).isTrue();
assertThat(NumberUtils.isCreatable("10.0d")).isTrue();
assertThat(NumberUtils.isCreatable("1000L")).isTrue();
assertThat(NumberUtils.isCreatable("0xFF")).isTrue();
assertThat(NumberUtils.isCreatable("07")).isTrue();
assertThat(NumberUtils.isCreatable("2.99e+8")).isTrue();

assertThat(NumberUtils.isCreatable(null)).isFalse();
assertThat(NumberUtils.isCreatable("")).isFalse();
assertThat(NumberUtils.isCreatable("abc")).isFalse();
assertThat(NumberUtils.isCreatable(" 22 ")).isFalse();
assertThat(NumberUtils.isCreatable("09")).isFalse();
```


Note that we're getting *true* assertions for hexadecimal numbers, octal numbers and scientific notations in lines 6, 7 and 8, respectively.

Also, on line 14, the string *"09"* returns *false* because the preceding *"0"* indicates that this is an octal number, and *"09"* is not a valid octal number.

For every input that returns *true* with this method, we can use *NumberUtils.createNumber(String)* (https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/math/NumberUtils.html#createNumber-java.lang.String-), which will give us the valid number.

5.2. *NumberUtils.isParsable(String)*

The *NumberUtils.isParsable(String) (https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/math/NumberUtils.html#isParsable-java.lang.String-)* method checks whether the given *String* is parsable or not.

Parsable numbers are those that are parsed successfully by any parse method like *Integer.parseInt(String)*, *Long.parseLong(String)*, *Float.parseFloat(String)* or *Double.parseDouble(String)*.

Unlike *NumberUtils.isCreatable()*, this method won't accept hexadecimal numbers, scientific notations, or strings ending with any type of qualifier like *'f'*, *'F'*, *'d'*, *'D'*, *'l'* or *'L'*.

Let's look at some affirmations:

```
assertThat(NumberUtils.isParsable("22")).isTrue();
assertThat(NumberUtils.isParsable("-23")).isTrue();
assertThat(NumberUtils.isParsable("2.2")).isTrue();
assertThat(NumberUtils.isParsable("09")).isTrue();

assertThat(NumberUtils.isParsable(null)).isFalse();
assertThat(NumberUtils.isParsable("")).isFalse();
assertThat(NumberUtils.isParsable("6.2f")).isFalse();
assertThat(NumberUtils.isParsable("9.8d")).isFalse();
assertThat(NumberUtils.isParsable("22L")).isFalse();
assertThat(NumberUtils.isParsable("0xFF")).isFalse();
assertThat(NumberUtils.isParsable("2.99e+8")).isFalse();
```

On line 4, unlike *NumberUtils.isCreatable()*, the number starting with string *"0"* isn't considered an octal number, but a normal decimal number, and therefore it returns true.

We can use this method as a replacement for what we did in section 3, where we're trying to parse a number and checking for an error.

5.3. *StringUtils.isNumeric(CharSequence)*

The method *StringUtils.isNumeric(CharSequence)* *(https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html#isNumeric-java.lang.CharSequence-)* checks strictly for Unicode digits. This means:

1. Any digits from any language that is a Unicode digit is acceptable

2. Since a decimal point is not considered as a Unicode digit, it's not valid
3. Leading signs (either positive or negative) are also not acceptable

Now let's see this method in action:

```
assertThat(StringUtils.isNumeric("123")).isTrue();
assertThat(StringUtils.isNumeric("١٢٣")).isTrue();
assertThat(StringUtils.isNumeric("१२३")).isTrue();

assertThat(StringUtils.isNumeric(null)).isFalse();
assertThat(StringUtils.isNumeric("")).isFalse();
assertThat(StringUtils.isNumeric("  ")).isFalse();
assertThat(StringUtils.isNumeric("12 3")).isFalse();
assertThat(StringUtils.isNumeric("ab2c")).isFalse();
assertThat(StringUtils.isNumeric("12.3")).isFalse();
assertThat(StringUtils.isNumeric("-123")).isFalse();
```

Note that the input parameters in lines 2 and 3 are representing numbers *123* in Arabic and Devanagari, respectively. Since they're valid Unicode digits, this method returns *true* on them.

5.4. *StringUtils.isNumericSpace(CharSequence)*

The *StringUtils.isNumericSpace(CharSequence)* (https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html#isNumericSpace-java.lang.CharSequence-) checks strictly for Unicode digits and/or space. This is the same as *StringUtils.isNumeric()* except that it also accepts spaces, and not only leading and trailing spaces, but also if they're in between numbers:

```
assertThat(StringUtils.isNumericSpace("123")).isTrue();
assertThat(StringUtils.isNumericSpace("١٢٣")).isTrue();
assertThat(StringUtils.isNumericSpace("")).isTrue();
assertThat(StringUtils.isNumericSpace("  ")).isTrue();
assertThat(StringUtils.isNumericSpace("12 3")).isTrue();

assertThat(StringUtils.isNumericSpace(null)).isFalse();
assertThat(StringUtils.isNumericSpace("ab2c")).isFalse();
assertThat(StringUtils.isNumericSpace("12.3")).isFalse();
assertThat(StringUtils.isNumericSpace("-123")).isFalse();
```

6. Benchmarks

Before we conclude this article, let's go through some benchmark results to help us to analyze which of the above mentioned methods are best for our use-case.

6.1. Simple Benchmark

First, we take a simple approach. We pick one string value – for our test we use *Integer.MAX_VALUE*. That value will then be tested against all our implementations:

```
Benchmark                                    Mode  Cnt    Score
Error  Units
Benchmarking.usingCoreJava                   avgt   20   57.241 ±
0.792  ns/op
Benchmarking.usingNumberUtils_isCreatable    avgt   20   26.711 ±
1.110  ns/op
Benchmarking.usingNumberUtils_isParsable     avgt   20   46.577 ±
1.973  ns/op
Benchmarking.usingRegularExpressions         avgt   20  101.580 ±
4.244  ns/op
Benchmarking.usingStringUtils_isNumeric      avgt   20   35.885 ±
1.691  ns/op
Benchmarking.usingStringUtils_isNumericSpace avgt   20   31.979 ±
1.393  ns/op
```

As we can see, the most costly operations are regular expressions. After that is our core Java-based solution.

Moreover, note that the operations using the Apache Commons library are by-and-large the same.

6.2. Enhanced Benchmark

Let's use a more diverse set of tests for a more representative benchmark:

- 95 values are numeric (0-94 and *Integer.MAX_VALUE*)
- 3 contain numbers but are still malformatted — '*x0*', '*0.*.005', and '*–11*'
- 1 contains only text
- 1 is a *null*

Upon executing the same tests, we'll see the results:

```
Benchmark                                    Mode  Cnt      Score
Error  Units
Benchmarking.usingCoreJava                   avgt   20  10162.872 ±
798.387  ns/op
Benchmarking.usingNumberUtils_isCreatable    avgt   20   1703.243 ±
108.244  ns/op
Benchmarking.usingNumberUtils_isParsable     avgt   20   1589.915 ±
203.052  ns/op
Benchmarking.usingRegularExpressions         avgt   20   7168.761 ±
344.597  ns/op
Benchmarking.usingStringUtils_isNumeric      avgt   20   1071.753 ±
8.657  ns/op
Benchmarking.usingStringUtils_isNumericSpace avgt   20   1157.722 ±
24.139  ns/op
```


The most important difference is that two of our tests, the regular expressions solution and the core Java-based solution, have traded places.

From this result, we learn that throwing and handling of the *NumberFormatException*, which occurs in only 5% of the cases, has a relatively big impact on the overall performance. So we can conclude that the optimal solution depends on our expected input.

Also, we can safely conclude that we should use the methods from the Commons library or a method implemented similarly for optimal performance.

7. Conclusion

In this article, we explored different ways to find if a *String* is numeric or not. We looked at both solutions – built-in methods and external libraries.

As always, the implementation of all examples and code snippets given above, including the code used to perform benchmarks, can be found over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-string-operations).

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)